

Spring Boot JPA

实践课程需求说明

说明

- 今天课程属于Spring Boot 在项目中最常用的基础，因此需要牢固掌握
- 重点在于 掌握Repository的构建，与使用，如果实现复杂查询，多数据源建议通过框架来处理，在微服务中不推荐使用

需求

- 构建一个JPA程序
- 实现多表联合查询
- 实现事物
- 实现自定义SQL

JPA 由来

ORM 框架能够将 Java 对象映射到关系数据库中，能够直接持久化复杂的 Java 对象。ORM 框架的出现，可以让开发者从数据库编程中解脱出来，把更多的精力放在了业务模型与业务逻辑上。目前比较流行的 ORM 框架有 Hibernate、MyBatis、TopLink、Spring JDBC

在 JPA 规范之前，由于没有官方的标准，使得各 ORM 框架之间的 API 差别很大，使用了某种 ORM 框架的系统会严重受制于该 ORM 的标准。基于此，Sun 引入新的 JPA ORM，主要的原因有：其一，简化现有 Java EE 和 Java SE 应用开发工作；其二，Sun 希望整合 ORM 技术，实现统一的 API 调用接口。

JPA 是什么

JPA（Java Persistence API）是 Sun 官方提出的 Java 持久化规范。它为 Java 开发人员提供了一种对象 / 关联映射工具来管理 Java 应用中的关系数据。它的出现主要是为了简化现有的持久化开发工作和整合 ORM 技术，结束现在 Hibernate、TopLink、JDO 等 ORM 框架各自为营的局面。

Spring Data JPA

Spring Data JPA 是 Spring 基于 ORM 框架、JPA 规范的基础上封装的一套 JPA 应用框架，可以让开发者用极简的代码即可实现对数据的访问和操作。它提供了包括增、删、改、查等在内的常用功能，且易于扩展，学习并使用 Spring Data JPA 可以极大提高开发效率。Spring Data JPA 其实就是 Spring 基于 Hibernate 之上构建的 JPA 使用解决方案，方便在 Spring Boot 项目中使用 JPA 技术。

Spring Data JPA 让我们解脱了 DAO 层的操作，基本上所有 CRUD 都可以依赖于它实现。

添加配置文件

```
spring.datasource.url=jdbc:mysql://localhost:3306/test?serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8&useSSL=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.properties.hibernate.hbm2ddl.auto=create
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
#SQL 输出
spring.jpa.show-sql=true
#format 一下 SQL 进行输出
spring.jpa.properties.hibernate.format_sql=true
```

hibernate.hbm2ddl.auto 参数的作用主要用于：自动创建、更新、验证数据库表结构，有四个值。

- create：每次加载 Hibernate 时都会删除上一次生成的表，然后根据 model 类再重新来生成新表，哪怕两次没有任何改变也要这样执行，这就是导致数据库表数据丢失的一个重要原因。
- create-drop：每次加载 Hibernate 时根据 model 类生成表，但是 sessionFactory 一关闭，表就自动删除。
- update：最常用的属性，第一次加载 Hibernate 时根据 model 类会自动建立起表的结构（前提是先建立好数据库），以后加载 Hibernate 时根据 model 类自动更新表结构，即使表结构改变了，但表中的行仍然存在，不会删除以前的行。要注意的是当部署到服务器后，表结构是不会被马上建立起来的，是要等应用第一次运行起来后才会。
- validate：每次加载 Hibernate 时，验证创建数据库表结构，只会和数据库中的表进行比较，不会创建新表，但是会插入新值。
其中：

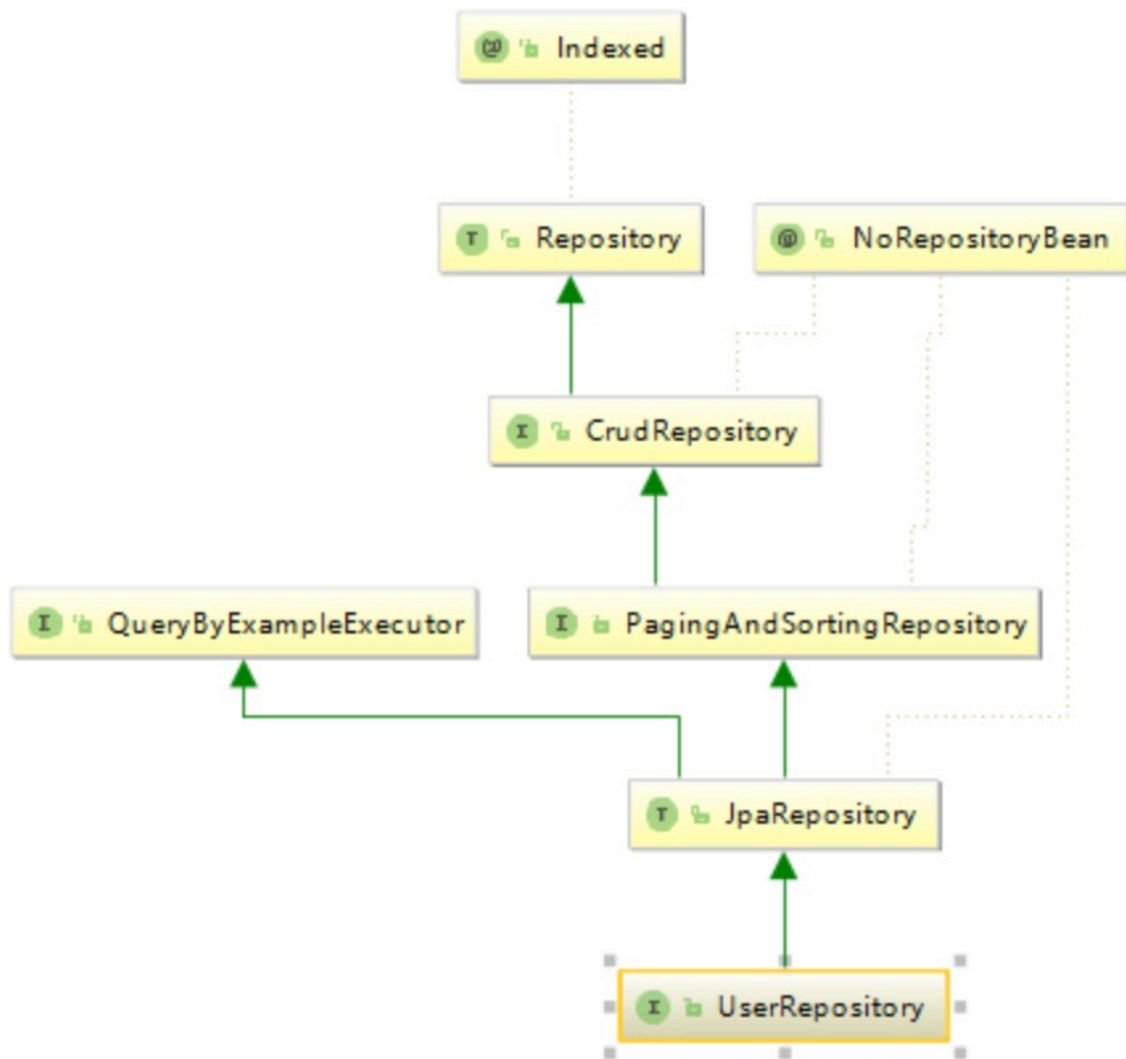
- dialect 主要是指指定生成表名的存储引擎为 InnoDB
- show-sql 是否在日志中打印出自动生成的 SQL，方便调试的时候查看

实体类注解

- @Entity(name="EntityName") 必须，用来标注一个数据库对应的实体，数据库中创建的表名默认和类名一致。其中，name 为可选，对应数据库中一个表，使用此注解标记 Pojo 是一个 JPA 实体。
- @Table(name="", catalog="", schema="") 可选，用来标注一个数据库对应的实体，数据库中创建的表名默认和类名一致。通常和 @Entity 配合使用，只能标注在实体的 class 定义处，表示实体对应的数据库表的信息。
- @Id 必须，@Id 定义了映射到数据库表的主键的属性，一个实体只能有一个属性被映射为主键。
- @GeneratedValue(strategy=GenerationType, generator="") 可选，strategy: 表示主键生成策略，有 AUTO、IDENTITY、SEQUENCE 和 TABLE 4 种，分别表示让 ORM 框架自动选择，generator: 表示主键生成器的名称。
- @Column(name = "user_code", nullable = false, length=32) 可选，@Column 描述了数据库表中该字段的详细定义，这对于根据 JPA 注解生成数据库表结构的工具。name: 表示数据库表中该字段的名称，默认情形属性名称一致；nullable: 表示该字段是否允许为 null，默认为 true；unique: 表示该字段是否是唯一标识，默认为 false；length: 表示该字段的大小，仅对 String 类型的字段有效。
- @Transient 可选，@Transient 表示该属性并非一个到数据库表的字段的映射，ORM 框架将忽略该属性。
- @Enumerated 可选，使用枚举的时候，我们希望数据库中存储的是枚举对应的 String 类型，而不是枚举的索引值，需要在属性上面添加 @Enumerated(EnumType.STRING) 注解。

Repository 构建 重点

- 创建的 Repository 只要继承 JpaRepository 即可，就会帮我们自动生成很多内置方法。
- 另外还有一个功能非常实用，可以根据方法名自动生产 SQL，比如 findByUserName 会自动生产一个以 userName 为参数的查询方法，比如 findAll 会自动查询表里面的所有数据等。



- JpaRepository 继承 PagingAndSortingRepository 和 QueryByExampleExecutor, PagingAndSortingRepository 类主要负责排序和分页内容, QueryByExampleExecutor 提供了很多示例的查询方法, 如下:

```

public interface QueryByExampleExecutor<T> {
    <S extends T> S findOne(Example<S> example); //根据“实例”查找一个对象
    <S extends T> Iterable<S> findAll(Example<S> example); //根据“实例”查找一批对象
    <S extends T> Iterable<S> findAll(Example<S> example, Sort sort); //根据“实例”查找一批对象, 且排序
    <S extends T> Page<S> findAll(Example<S> example, Pageable pageable); //根据“实例”查找一批对象, 且排序和分页
    <S extends T> long count(Example<S> example); //根据“实例”查找, 返回符合条件的对象个数
    <S extends T> boolean exists(Example<S> example); //根据“实例”判断是否有符合条件的对象
}

```

- 因此，继承 JpaRepository 的会自动拥有上述这些方法和排序、分页功能。查看源码我们发现 PagingAndSortingRepository 又继承了 CrudRepository。CrudRepository 的源码如下：

```
@NoRepositoryBean
public interface CrudRepository<T,ID> extends Repository<T,ID> {
    <S extends T> S save(S entity);
    <S extends T> Iterable<S> saveAll(Iterable<S> entities);
    Optional<T> findById(ID id);
    boolean existsById(ID id);
    Iterable<T> findAll();
    Iterable<T> findAllById(Iterable<ID> ids);
    long count();
    void deleteById(ID id);
    void delete(T entity);
    void deleteAll(Iterable<? extends T> entities);
    void deleteAll();
}
```

- 从 CrudRepository 的源码可以看出 CrudRepository 内置了我们最常用的增、删、改、查的方法，方便我们去使用，因为 JpaRepository 继承了 PagingAndSortingRepository，PagingAndSortingRepository 继承了 CrudRepository，所以继承 JpaRepository 的类也默认拥有了上述方法。
- 因此使用 JPA 操作数据库时，只需要构建的 Repository 继承了 JpaRepository，就会拥有了很多常用的数据库操作方法。

基本查询

- 预生成方法

继承JpaRepository

- 自定义查询

Spring Data JPA 可以根据接口方法名来实现数据库操作，主要的语法是 findXXBy、readAXXBy、queryXXBy、countXXBy、getXXBy 后面跟属性名称，利用这个功能仅需要在定义的 Repository 中添加对应的方法名即可，使用时 Spring Boot 会自动帮我们实现，示例如下。

根据用户名查询用户：

```
User findByUserName(String userName);
```

也可以加一些关键字 And、or:

```
User findByUserNameOrEmail(String username,String email);
```

修改、删除、统计也是类似语法:

```
Long deleteById(Long id);  
Long countByUserName(String userName)
```

基本上 SQL 体系中的关键词都可以使用, 如 LIKE 、IgnoreCase、OrderBy:

```
List<User> findByEmailLike(String email);  
  
User findByUserNameIgnoreCase(String userName);  
  
List<User> findByUserNameOrderByEmailDesc(String email);
```

具体的关键字, 使用方法和生产成 SQL 如下表所示

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstnames, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age ≤ ?1
GreaterTha		

n	findByAgeGreaterThan	... where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection age)	... where x.age not in ?1
TRUE	findByActiveTrue()	... where x.active = true
FALSE	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

自定义查询

- HQL 在 SQL 的查询方法上面使用 @Query 注解，在注解内写 Hql 来查询内容。

```
@Query("select u from User u")
Page<User> findALL(Pageable pageable);
```

- 原生 SQL

```
@Query("select * from user u where u.nick_name = ?1", nativeQuery = true)
Page<User> findByNickName(String nickName, Pageable pageable);
```

@Query 上面的 1 代表的是方法参数里面的顺序，如果有多个参数也可以按照这个方式添加 1、2、3

- 还可以使用 @Param 来支持

```
@Query("select u from User u where u.nickName = :nickName")
Page<User> findByNickName(@Param("nickName") String nickName, Pageable pageable);
```

- 涉及到删除和修改需要加上 @Modifying，也可以根据需要添加 @Transactional 对事务的支持、操作超时设置

```
@Transactional(timeout = 10)
@Modifying
@Query("update User set userName = ?1 where id = ?2")
int modifyById(String userName, Long id);

@Transactional
@Modifying
@Query("delete from User where id = ?1")
void deleteById(Long id);
```

- 使用已命名的查询

除了使用 @Query 注解外，还可以预先定义好一些查询，并为其命名，然后再 Repository 中添加相同命名的方法。

定义命名的 Query：

```
@Entity
@NamedQueries({
```



```
@NamedQuery(name = "User.findByPassWord", query = "select u from User u where  
u.passWord = ?1"),  
@NamedQuery(name = "User.findByNickName", query = "select u from User u where  
u.nickName = ?1"),  
})  
public class User {  
.....  
}
```

通过 @NamedQueries 注解可以定义多个命名 Query，@NamedQuery 的 name 属性定义了 Query 的名称，注意加上 Entity 名称 . 作为前缀，query 属性定义查询语句。

定义对应的方法：

```
List<User> findByPassWord(String passWord);  
List<User> findByNickName(String nickName);
```

- Query 查找策略
- 通过方法名自动创建 Query
- 通过 @Query 注解实现自定义 Query
- 通过 @NamedQuery 注解来定义 Query

Spring Data JPA 如何来查找这些 Query 呢？

- 通过配置 @EnableJpaRepositories 的 queryLookupStrategy 属性来配置 Query 查找策略，有如下定义。
- CREATE：尝试从查询方法名构造特定于存储的查询。一般的方法是从方法名中删除一组已知的前缀，并解析方法的其余部分。
- USE_DECLARED_QUERY：尝试查找已声明的查询，如果找不到，则抛出异常。查询可以通过某个地方的注释定义，也可以通过其他方式声明。
- CREATE_IF_NOT_FOUND（默认）：CREATE 和 USE_DECLARED_QUERY 的组合，它首先查找一个已声明的查询，如果没有找到已声明的查询，它将创建一个自定义方法基于名称的查询。它允许通过方法名进行快速查询定义，还可以根据需要引入声明的查询来定制这些查询调优。

一般情况下使用默认配置即可，如果确定项目 Query 的具体定义方式，可以更改上述配置，例如，全部使用 @Query 来定义查询，又或者全部使用命名的查询。

分页查询

Spring Data JPA 已经帮我们内置了分页功能，在查询的方法中，需要传入参数 Pageable，当查询中有多个参数的时候 Pageable 建议作为最后一个参数传入。

```
@Query("select u from User u")
Page<User> findALL(Pageable pageable);

Page<User> findByNickName(String nickName, Pageable pageable);
```

- Pageable 是 Spring 封装的分页实现类，使用的时候需要传入页数、每页条数和排序规则，Page 是 Spring 封装的分页对象，封装了总页数、分页数据等。返回对象除使用 Page 外，还可以使用 Slice 作为返回值。

```
Slice<User> findByNickNameAndEmail(String nickName, String email, Pageable pageable)
;
```

Page 和 Slice 的区别如下

- Page 接口继承自 Slice 接口，而 Slice 继承自 Iterable 接口。
- Page 接口扩展了 Slice 接口，添加了获取总页数和元素总数量的方法，因此，返回 Page 接口时，必须执行两条 SQL，一条复杂查询分页数据，另一条负责统计数据数量。
- 返回 Slice 结果时，查询的 SQL 只会有查询分页数据这一条，不统计数据数量。
- 用途不一样：Slice 不需要知道总页数、总数据量，只需要知道是否有下一页、上一页，是否是首页、尾页等，比如前端滑动加载一页可用；而 Page 知道总页数、总数据量，可以用于展示具体的页数信息，比如后台分页查询。

```
@Test
public void testPageQuery() {
    int page=1,size=2;
    Sort sort = new Sort(Sort.Direction.DISC, "id");
    Pageable pageable = PageRequest.of(page, size, sort);
    userRepository.findAll(pageable);
    userRepository.findByNickName("aa", pageable);
}
```

- Sort，控制分页数据的排序，可以选择升序和降序。
- PageRequest，控制分页的辅助类，可以设置页码、每页的数据条数、排序等。还有一些更简洁的方式来排序和分页查询，如下。

限制查询

有时候我们只需要查询前 N 个元素，或者只取前一个实体。

```
User findFirstByOrderByLastNameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

复杂查询

- 我们可以通过 AND 或者 OR 等连接词来不断拼接属性来构建多条件查询，但如果参数大于 6 个时，方法名就会变得非常的长，并且还不能解决动态多条件查询的场景。到这里就需要给大家介绍另外一个利器 **JpaSpecificationExecutor** 了。
- `JpaSpecificationExecutor` 是 JPA 2.0 提供的 Criteria API 的使用封装，可以用于动态生成 Query 来满足我们业务中的各种复杂场景。Spring Data JPA 为我们提供了 `JpaSpecificationExecutor` 接口，只要简单实现 `toPredicate` 方法就可以实现复杂的查询。

`JpaSpecificationExecutor` 的源码：

```
public interface JpaSpecificationExecutor<T> {
    //根据 Specification 条件查询单个对象，注意的是，如果条件能查出来多个会报错
    T findOne(@Nullable Specification<T> spec);
    //根据 Specification 条件查询 List 结果
    List<T> findAll(@Nullable Specification<T> spec);
    //根据 Specification 条件，分页查询
    Page<T> findAll(@Nullable Specification<T> spec, Pageable pageable);
    //根据 Specification 条件，带排序的查询结果
    List<T> findAll(@Nullable Specification<T> spec, Sort sort);
    //根据 Specification 条件，查询数量
    long count(@Nullable Specification<T> spec);
}
```

- Root root，代表了可以查询和操作的实体对象的根，开一个通过 `get("属性名")` 来获取对应的值。
- CriteriaQuery query，代表一个 specific 的顶层查询对象，它包含着查询的各个部分，比如

select、from、where、group by、order by 等。

- CriteriaBuilder cb，来构建 CritiaQuery 的构建器对象，其实就相当于条件或者是条件组合，并以 Predicate 的形式返回。

下面的使用案例中会报错这几个对象的使用。

- 首先定义一个 UserDetails 对象，作为演示的数据模型。

```
@Entity
public class UserDetails {
    @Id
    @GeneratedValue
    private Long id;
    @Column(nullable = false, unique = true)
    private Long userId;
    private Integer age;
    private String realName;
    private String status;
    private String hobby;
    private String introduction;
    private String lastLoginIp;
}
```

- 创建 UserDetails 对应的 Repository：

```
public interface UserDetailsRepository extends JpaSpecificationExecutor<UserDetail>,
JpaRepository<UserDetail, Long> {
}
```

- 定义一个查询 Page 的接口：

```
public interface UserDetailsService {
    public Page<UserDetail> findByCondition(UserDetailParam detailParam, Pageable pageable);
}
```

- 在 UserDetailsServiceImpl 中，我们来演示 JpaSpecificationExecutor 的具体使用。

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService{

    @Resource
```

```

private UserDetailsRepository userDetailsRepository;

@Override
public Page<UserDetail> findByCondition(UserDetailParam detailParam, Pageable p
ageable){

    return userDetailsRepository.findAll((root, query, cb) -> {
        List<Predicate> predicates = new ArrayList<Predicate>();
        //equal 示例
        if (!StringUtils.isEmpty(detailParam.getIntroduction())){
            predicates.add(cb.equal(root.get("introduction"),detailParam.getInt
roduction()));
        }
        //like 示例
        if (!StringUtils.isEmpty(detailParam.getRealName())){
            predicates.add(cb.like(root.get("realName"), "%"+detailParam.getReal
Name()+"%"));
        }
        //between 示例
        if (detailParam.getMinAge()!=null && detailParam.getMaxAge()!=null) {
            Predicate agePredicate = cb.between(root.get("age"), detailParam.ge
tMinAge(), detailParam.getMaxAge());
            predicates.add(agePredicate);
        }
        //greaterThan 大于等于示例
        if (detailParam.getMinAge()!=null){
            predicates.add(cb.greaterThan(root.get("age"),detailParam.getMinAge
()));
        }
        return query.where(predicates.toArray(new Predicate[predicates.size()])
).getRestriction();
    }, pageable);

}
}

```

上面的示例是根据不同条件来动态查询 UserDetail 分页数据，UserDetailParam 是参数的封装，示例中使用了常用的大于、like、等于等示例，根据这个思路我们可以不断扩展完成更复杂的动态 SQL 查询。

使用时只需要将 UserDetailsService 注入调用相关方法即可：

```

@RunWith(SpringRunner.class)
@SpringBootTest
public class JpaSpecificationTests {

```

```

@Resource
private UserDetailsServiceImpl userDetailsServiceImpl;

@Test
public void testFindByCondition() {
    int page=0,size=10;
    Sort sort = new Sort(Sort.Direction.DESC, "id");
    Pageable pageable = PageRequest.of(page, size, sort);
    UserDetailsParam param=new UserDetailsParam();
    param.setIntroduction("程序员");
    param.setMinAge(10);
    param.setMaxAge(30);
    Page<UserDetail> page1=userDetailsServiceImpl.findByCondition(param,pageable);
    for (UserDetail userDetails:page1){
        System.out.println("userDetail: "+userDetails.toString());
    }
}
}

```

多表查询

多表查询在 Spring Data JPA 中有两种实现方式，**第一种是利用 *Hibernate* 的级联查询来实现，第二种是创建一个结果集的接口来接收连表查询后的结果**，这里主要介绍第二种方式。

我们还是使用上面的 `UserDetail` 作为数据模型来使用，定义一个结果集的接口类，接口类的内容来自于用户表和用户详情表。

```

public interface UserInfo {
    String getUsername();
    String getEmail();
    String getAddress();
    String getHobby();
}

```

- 在运行中 Spring 会给接口（`UserInfo`）自动生产一个代理类来接收返回的结果，代码中使用 `getXX` 的形式来获取。
- 在 `UserDetailRepository` 中添加查询的方法，返回类型设置为 `UserInfo`：

```

@Query("select u.userName as userName, u.email as email, d.introduction as introduction , d.hobby as hobby from User u , UserDetails d " +
"where u.id=d.userId and d.hobby = ?1 ")

```

```
List<UserInfo> findUserInfo(String hobby);
```

特别注意这里的 **SQL** 是 **HQL**，需要写类的名和属性，这块很容易出错。

测试验证：

```
@Test
public void testUserInfo() {
    List<UserInfo> userInfos=userDetailRepository.findUserInfo("钓鱼");
    for (UserInfo userInfo:userInfos){
        System.out.println("userInfo: "+userInfo.getUserName()+"-"+userInfo.getEmail()+"-
        "+userInfo.getHobby()+"-"+userInfo.getIntroduction());
    }
}
```

运行测试方法后返回：

userInfo: aa-aa@126.com-钓鱼-程序员

- 证明关联查询成功，最后的返回结果来自于两个表，按照这个思路可以进行三个或者更多表的关联查询。

Spring Data JPA 多数据源的使用

项目中使用多个数据源在以往工作中比较常见，微服务架构中不建议一个项目使用多个数据源。在微服务架构下，一个微服务拥有自己独立的一个数据库，如果此微服务要使用其他数据库的数据，需要调用对应库的微服务接口来调用，而不是在一个项目中连接使用多个数据库，这样微服务更独立、更容易水平扩展。

Spring Data JPA 使用多数据源的整体思路是，配置不同的数据源，在启动时分别加载多个数据源配置，并且注入到不同的 repository 中。这样不同的 repository 包就有不同的数据源，使用时注入对应包下的 repository，就会使用对应数据源的操作。

多数据源的支持

配置 Spring Data JPA 对多数据源的使用，一般分为以下几步：

- 创建数据库 test1 和 test2 配置多数据源
- 不同源的 repository 放入不同包路径
- 声明不同的包路径下使用不同的数据源、事务支持

- 不同的包路径下创建对应的 repository
- 测试使用

配置两个数据源：

```
spring.datasource.primary.jdbc-url=jdbc:mysql://localhost:3306/test1?serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8&useSSL=true
```

```
spring.datasource.primary.username=root
```

```
spring.datasource.primary.password=root
```

```
spring.datasource.primary.driver-class-name=com.mysql.cj.jdbc.Driver
```

```
spring.datasource.secondary.jdbc-url=jdbc:mysql://localhost:3306/test2?serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8&useSSL=true
```

```
spring.datasource.secondary.username=root
```

```
spring.datasource.secondary.password=root
```

```
spring.datasource.secondary.driver-class-name=com.mysql.cj.jdbc.Driver
```

#SQL 输出

```
spring.jpa.show-sql=true
```

```
spring.jpa.properties.hibernate.hbm2ddl.auto=create
```

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
```

#format 一下 SQL 进行输出

```
spring.jpa.properties.hibernate.format_sql=true
```